מערכות הפעלה

מרצה: ערן טרומר

סמסטר א' תשע"ב

הרצאה 7

תכנות תהליכים בו-זמניים

מטרות בתכנות בו-זמני

- רצון לנצל מספר מעבדים פיזיים במחשב
- רצון לנצל את המעבד בזמן שהתוכנית ממתינה להתקן איטי, כגון קריאה מהדיסק
- מתן אשליה למשתמש שמספר פעולות קורות במקביל, כמו טיפול בקלט בזמן שהתוכנית ממשיכה להציג אנימציה או קול ברצף, גם אם יש מעבד אחד
- מימוש שרתים שבהם כל לקוח מטופל על ידי חוט נפרד
- פרט לראשונה, דוגמאות למודולריצזיה טמפורלית : רצון להפריד מודולים שיכולים לרוץ בכל מיני סדרי ביצוע

תיאום

- כל חוט מבצע פעולות אחרות
- אם ניתן לבצע את הפעולות של חוטים שונים במקביל או בכל סדר שהוא, שגר ושכח
- ברוב המקרים, יש לכפות אילוצים על סדר הפעולות על מנת להבטיח נכונות
- דוגמה: שני חוטים שכל אחד מבצע i=i+1
- אם הראשון קורא את i לאוגר, מקדם את האוגר, השני קורא את i לאוגר, מקדם את האוגר, הראשון כותב את i לזיכרון, השני כותב את תוכן האוגר ל-i, המשתנה קודם ב-1 ולא ב-2

מנעולים

- מנעול (lock / mutex) הוא עצם שמבטיח **מניעה הדדית**
- שני מצבים: חפשי ונעול
- ניסיון נעילה של מנעול חפשי מצליח ושגרת הנעילה חוזרת
- לחוט מותר לשחרר רק חוט שנעול על ידו
- ניסיון נעילה של מנעול נעול ממתין לשחררו
- חוט שבידו מנעול (נעול) מונע מחוטים אחרים לנעול אותו :

```
lock(m)
i=i+1
unlock(m)
```

אירועים

- משתנה תנאי (condition variable / events) הוא מנגנון שבעזרתו חוט יכול לחכות לקבלת הודעה על אירוע מחוט אחר
- המנגנון שיתואר תקף בלינוקס/יוניקס, בחלונות מנגנון דומה אך שונה בפרטים
- **`wait(c,m)`** משחררת את המנעול **`m`** ונכנסת להמתנה לאירוע **`c`** <u>באופן אטומי</u>
- **`signal(c)`** משחררת את אחד החוטים שממתין ל-**`c`**
- **`broadcast(c)`** משחררת את כל החוטים שממתינים ל-**`c`**
- ל-**`signal`** ו- **`broadcast`** אין שום השפעה אם אין חוטים ממתינים
- אחרי התעוררות ולפני חזרה מ-**`wait`** המנעול ננעל שוב, אולי לאחר המתנה (לא אטומי)

דוגמה: קורא וכותב

```
forever
  lock(m)
  while (state==full)
    wait(c,m)
  כתוב לחוצץ
  state = full
  signal(c)
  unlock(m)
  פעולות אחרות
```

```
forever
  lock(m)
  while (state==empty)
    wait(c,m)
  קרא מהחוצץ
  state = empty
  signal(c)
  unlock(m)
  פעולות אחרות
```

למה while ולא if?

```
forever
  lock(m)
  while/if (state==full)
    wait(c,m)
  כתוב לחוצץ
  state = full
  signal(c)
  unlock(m)
  פעולות אחרות
```

- מבטיח שחוט לא יבצע פעולה אלא אם התנאי מתקיים
- שימוש ב-if היה עלול לגרום לביצוע הפעולה אם הודעה על האירוע לא תמיד מבטיחה שהתנאי מתקיים
- if נכון כאן, אבל עלול לגרום שגיאות בתוכניות מורכבות
- while מאפשר בדיקת נכונות מקומית, if מצריך בדיקה גלובלית

מספר קוראים וכותבים

- כאשר יש מספר חוטים שכותבים וקוראים, **signal** עלול להעיר חוט שהתנאי שהוא מחכה לו לא מתקיים (כותב מעיר כותב אחר)
- החוט שהתעורר יגלה שהתנאי לא מתקיים ויחזור לישון
- האירוע אבד וחוט שמחכה לתנאי שכן מתקיים עלול שלא להתעורר לעולם
- **broadcast** פותר את הבעיה כי כל הממתינים מתעוררים
- עלות גבוהה יותר מ-**signal**

```
forever
   lock(m)
   while (state==full)
      wait(c,m)
   כתוב לחוצץ
   state = full
   broadcast(c)
   unlock(m)
   פעולות אחרות
```

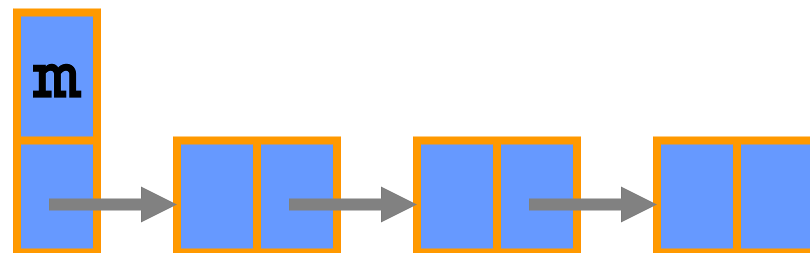
פתרון יותר יעיל עם 2 אירועים

```
forever
  lock(m)
  while (state==full)
    wait(c_e,m)
  כתוב לחוצץ
  state = full
  signal(c_f)
  unlock(m)
  פעולות אחרות
```

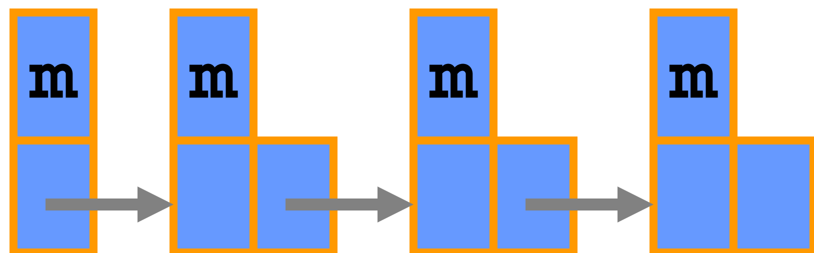
```
forever
  lock(m)
  while (state==empty)
    wait(c_f,m)
  קרא מהחוצץ
  state = empty
  signal(c_e)
  unlock(m)
  פעולות אחרות
```

יעיל יותר ונכון

גרעיניות



- מנעול אחד לרשימה שלמה
- חוט אחד לכל היותר מטפל ברשימה בזמן נתון
- נעילה אחת לכל פעולה על הרשימה



- מנעול לכל איבר ברשימה
- מספר חוטים יכולים לטפל ברשימה בו-זמנית
- נעילה בכל מעבר באיבר; יותר תקורה בגלל בו-זמניות

אין פתרון מיטבי גלובלי; הפתרון העדיף תלוי ביישום

קיפאון

חוט א':

```
lock(m2)
lock(m1)
unlock(m2)
unlock(m1)
```

חוט א':

```
lock(m1)
lock(m2)
unlock(m2)
unlock(m1)
```

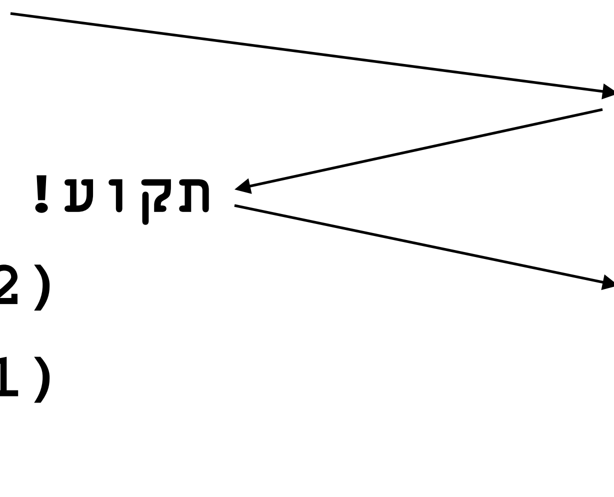
קיפאון: דוגמה לתזמון

חוט א':

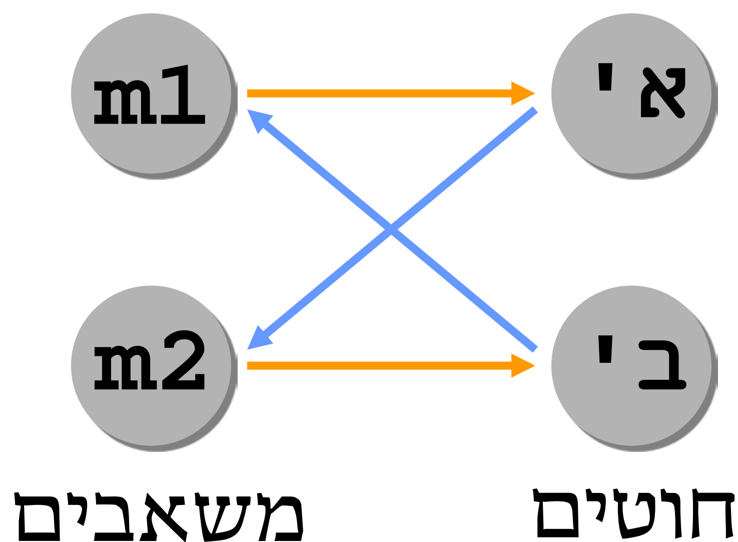
```
lock(m2)
lock(m1) תקוע!
unlock(m1)
unlock(m2)
```

חוט ב':

```
lock(m1)
lock(m2) תקוע!
unlock(m2)
unlock(m1)
```



מודל פורמלי: גרף דו-צדדי



חוט מחכה למשאב

חוט נועל משאב

מעגל מכוון ← קיפאון

הימנעות מקיפאון

- אם חוט נועל מנעול אחד לכל היותר בכל נקודת זמן
- אם ניתן להגדיר סדר חלקי על משאבים וחוט תמיד נועל לפי סדר:
  - סדר שכבות התוכנה אם בכל שכבה נועלים משאב אחד לכל היותר
  - סדר איברים ברשימה (לא דו-צדדית)
  - סדר איברים בעץ מהשורש לעלים או מהעלים לשורש
  - סדר שרירותי (כתובות של משאבים, למשל) אם ניתן לנעול בכל סדר
- לא תמיד קל להגדיר סדר ולהבטיח נעילה לפי סדר (למשל כאשר משתמשים בתור עדיפויות ממומש כ-heap)
- חוטים עלולים להיכנס לקיפאון לא רק בגלל שימוש במנעולים אלא גם בגלל שימוש במשאבים בלעדיים אחרים או אירועים

היחלצות מקיפאון

- מערכות מסוימות מסוגלות לזהות מעגל בגרף, להעיף את אחד החוטים, ועל ידי כך לשבור את הקיפאון
- מערכות מסדי נתונים מעיפות את אחד החוטים הקפואים או כאשר יש חשד לקיפאון (אי שחרור משאב תוך פרק זמן סביר). אם החוט התחיל לעדכן את מסד הנתונים, המצב הישן משוחזר.
- לא ישים במערכות הפעלה, כיון שבדרך כלל תוכנית לא מסוגלת להמשיך לרוץ כאשר אחד החוטים שלה עף, ולא ברור איך לשחזר מצבעיקבי.
- גם בחלונות וגם ביוניקס/לינוקס ניתן להגביל את זמן ההמתנה לאירוע (בחלונות גם למנעול)
- עדיף לכתוב תוכנית שלא יתכן בה קיפאון!

אבחון וניפוי קיפאון

- קיפאון הוא תקלה נפוצה בתוכניות בו-זמניות
- קיפאון הוא גם תקלה קלה לאבחון: החוטים נתקעים וניתן לבחון את הסיבה לקיפאון (המעגל בגרף) במנפה
- בגלל שהחוטים נתקעים הקיפאון קורה בדרך כלל קרוב לשגיאה בתוכנית, להבדיל מסוגי תקלות אחרות (בעיות זיכרון)
- ריצה ללא קיפאון אינה מבטיחה, כמובן, חיות בכל ריצה

הגינות: מנעולי קוראים/כותב

- להדגמת נושא ההגינות נממש מנעול קוראים/כותב
- צריך לאפשר או למנוע גישה למבנה נתונים (כמו מנעול)
- ארבע פעולות: נעל/שחרר לקריאה, נעל/שחרר לכתיבה
- מותר למספר בלתי מוגבל של קוראים לגשת למבנה הנתונים בו-זמנית בלי כותבים, או לכותב אחד בלבד בלי קוראים
- (ממומש כחלק מממשק החוטים של לינוקס ויוניקס)

פתרון פשוט

`rw.i` מייצג את מספר הקוראים (ערך חיובי) או הכותבים (ערך שלילי)

```
read_lock(rw)
  lock(rw.m)
  while (rw.i < 0)
    wait(rw.c,rw.m)
  rw.i++
  unlock(m)

read_unlock(rw)
  lock(rw.m)
  rw.i--
  if (rw.i == 0)
    signal(rw.c)
  unlock(rw.m)
```

```
write_lock(rw)
  lock(rw.m)
  while (rw.i != 0)
    wait(rw.c,rw.m)
  rw.i-- /* == -1 */
  unlock(m)

write_unlock(rw)
  lock(rw.m)
  rw.i++ /* == 0 */
  broadcast(rw.c)
  unlock(rw.m)
```

הפתרון אינו הוגן

- גם אם חוטים נועלים מנעולים ומקבלים אירועים לפי סדר הכניסה להמתנה (לא בהכרח נכון), הפתרון אינו הוגן
- כל זמן שיש קוראים, קוראים נוספים יצליחו לנעול וכותבים ימשיכו להמתין
- אם קוראים ממשיכים להגיע כל הזמן, הם ירעיבו את הכותבים
- יש מספר פתרונות הוגנים אבל צריך להגדיר את הדרישות
- למשל, המנעול עובר לסירוגין בין קוראים וכותב (אלא אם אין חוטים מהסוג השני), קוראים שנכנסים להמתנה בזמן שאחרים קוראים מחכים לסיבוב הבא

גם העצמים הבסיסיים אינן הוגנים

- מערכות הפעלה מספקות מנעולים ואירועים שמונעים הרעבה אבל אינם מבטיחים, בדרך כלל, שירות לפי סדר ההמתנה
- סיבה 1: הגינות מוחלטת קשה למימוש ועלולה לפגוע ביעילות
- סיבה 2: במחשב מרובה מעבדים מספר חוטים עשויים להיכנס להמתנה בו-זמנית ממש; אין סדר בין בקשות השירות
- כפי שראינו, גם אם מנעולים ואירועים היו הוגנים, תוכניות מורכבות עלולות להיות לא הוגנות ואף להרעיב חוטים
- בדרך כלל כדאי להניח שהעצמים הבסיסיים מונעים הרעבה ותו לא, ולממש הגינות בתוכנית עצמה אם יש צורך בכך

שיקולי יעילות

- עדיף להימנע מלהעיר חוטים שנכנסים מייד חזרה להמתנה (כלומר משימוש ב-broadcast במקום באירועים ספציפיים יותר)
- מערכת חלונות תומכת בשני סוגי מנעולים: משתני מניעה הדדית וקטעים קריטיים
  - משתנה מניעה הדדית ניתן לשימוש מתהליכים שונים ויש לו תקורה גבוהה של קריאת מערכת
  - בקטע קריטי ניתן להשתמש רק בחוטים של תהליך אחד ויש לו תקורה נמוכה, לפחות כאשר המנעול חפשי
- ביוניקס ולינוקס מנעולים ואירועים נגישים רק לתהליך בודד וניתן לממש אותם ביעילות

עדיפויות והיפוכן

- מערכות מסוימות תומכות בעדיפויות לחוטים
- לעיתים המשמעות של עדיפות היא שחוט בעדיפות גבוהה לעולם לא ימתין למעבד שמריץ חוט בעדיפות נמוכה
- היפוך עדיפויות (priority inversion):
  - חוט בעדיפות נמוכה נועל משאב
  - חוט בעדיפות בינונית רץ לאורך זמן רב
  - חוט בעדיפות גבוהה מתעורר, מבקש את המשאב הנעול, ונכנס להמתנה; למעשה החוט הזה ממתין שהחוט בעדיפות בינונית יסיים
- על מנת למנוע היפוך עדיפויות, חוט שנועל משאב צריך לרשת את העדיפות המירבית של כל מי שמחכה למשאב

אירועים בחלונות

- שני סוגי אירועים: איפוס ידני ואוטומטי
- שני הסוגים הם משתנים בינריים מיוחדים וניתן לקבוע את ערכם ל-1 או 0
- המתנה לאירוע מחכה שערכו יהיה 1 אם איננו כזה, ואינה ממתינה אם ערכו 1; אינה משחררת או נועלת כל מנעול
- השתחררות של חוט מהמתנה לאירוע אוטומטי מאפסת אותו, כך שרק חוט אחד מתעורר; האירוע זוכר שערכו 1 אם אין ממתינים ברגע שבו מעלים את ערכו
- אירוע ידני מאופס רק באופן מפורש

מימוש מנעולים

- קריאת מערכת במחשב עם מעבד אחד: משתנה בוליאני;
  - ```
    if (locked==true)
        suspend the thread & try later
    else
        locked=true // return from syscall; the
                    // thread acquired the lock
    ```
- אם יש יותר ממעבד אחד, זה לא עובד
- צריך לעצור את המעבדים או להשתמש במנגנון גישה יותר מתוחכם לזיכרון
    ```

Compare & Swap

- דוגמה למנגנון גישה כזה
- פקודת מכונה עם שלושה ארגומנטים: כתובת בזיכרון, ושני ערכים, ישן וחדש
- אם הכתובת מכילה את הערך הישן (לא נעול), הפקודה שמה בכתובת את הערך החדש (באופן אטומי); אחרת היא לא משנה את תוכן הזיכרון; הערך שהיה בכתובת מוחזר
- דורש מימוש ברמת המעבד ומערכת הזיכרון
- יש עוד מנגנונים אפשריים כאלה ויש להם עוד שימושים חוץ ממימוש מנעולים

מבני נתונים wait free

- נניח שכמה חוטים צריכים להשתמש מונה משותף
- הפעולה (inc(counter צריכה לקדם את המונה ב-1 ולהחזיר את ערכו החדש באופן אטומי
- קל למימוש בעזרת מנעול
- במימוש כזה, אם חוט שנועל את המונה יעוף, שאר החוטים יתקעו; החוט לא ישחרר לעולם את המנעול (אלא עם מערכת ההפעלה תגלה שהחוט עף, אבל זה דורש נעילה יקרה)
- קל לפתרון בדרך שהיא wait free (זה לא יקרה) בעזרת compare & swap; תרגיל

אתגר התכנות המקבילי

- תכנות מקבילי נהיה הכרחי למיצוי יכולות מעבדים מודרניים
  - חוק מור (מספר שערים במעבד) לעומת קצב שעון תקוע
  - זמן תגובה איטי למשאבים מחוץ למעבד, כולל זכרון ראשי
- בדרך כלל קשה הרבה יותר מכתיבת קוד סדרתי
- פתרונות בהתהוות:
  - ספריות קוד מקבילי
  - הפשטות ושפות תכנות המקלות על המתכנת להביע את תלויות הזמנים בין פיסות קוד ומידע
  - כלים אוטומטיים ליצירת קוד מקבילי (מהדרים חכמים וכו')
  - כלים אוטומטיים לבדיקה והוכחת נכונות